AFRL-RI-RS-TR-2010-9
Final Technical Report
January 2010

ACCOUNTABLE INFORMATION FLOW FOR JAVA-BASED WEB APPLICATIONS

Cornell University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2010-9 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
PHILIP B. TRICCA
Work Unit Manager

/s/
JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JANUARY 2010	Final	February 2008 – July 2009

4. TITLE AND SUBTITLE

ACCOUNTABLE INFORMATION FLOW FOR JAVA-BASED WEB APPLICATIONS

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-08-2-0079

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Andrew Myers

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
14

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
373 Pine Tree Road
Ithaca NY 14853

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIEBB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2010-9

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-5332

13. SUPPLEMENTARY NOTES

14. ABSTRACT

To enforce accountability of information in web applications, it is necessary to track and control information flows. Information flow control ensures that information affected by some source can be attributed to that source. This research explored new ways to control information flow in web applications by extending prior work on the JIF programming language. Several technical innovations made it possible to apply language-based information flow control in every tier of a web application: at the application server (in the SIF system), at the web browser (in the Swift system), and in the persistent store (in the Fabric system).

Several peer-reviewed publications were produced, some appearing in highly competitive publication venues. In addition, most of the software produced under the auspices of this project, including the SIF and Swift systems, has been publicly released, along with manuals and tutorials explaining how to use them to build web applications.

15. SUBJECT TERMS
Accountability, security, programming languages, web applications

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Philip B. Tricca
U	U	U	UU	26	19b. TELEPHONE NUMBER (*Include area code*) N/A

Abstract

This funded research explored language-based methods for enforcing accountability in web applications. Web applications are an important target for improving security because they are so widely used and are also very vulnerable. Better technology is needed for building web applications that are secure and that use information securely and accountably.

To enforce accountability of information in web applications, it is necessary to track and control information flows. Information flow control ensures that information affected by some source can be attributed to that source. This research explored new ways to control information flow in web applications by extending prior work on the Jif programming language. Language-based methods have the advantage that they can track information flow precisely, with fine granularity and low overhead. Jif combines compile-time and run-time methods to control information flow, enforcing confidentiality and integrity policies. Several technical innovations made it possible to apply language-based information flow control in every tier of a web application: at the application server (in the SIF system), at the web browser (in the Swift system), and in the persistent store (in the Fabric system). Each of these new systems provides strong, end-to-end enforcement of policies that can be used to ensure accountability.

Several peer-reviewed publications were produced, some appearing in highly competitive publication venues. In addition, most of the software produced under the auspices of this project, including the SIF and Swift systems, has been publicly released, along with manuals and tutorials explaining how to use them to build web applications.

Contents

1 Introduction **1**
- 1.1 The SIF web servlet architecture . . . 1
- 1.2 Automatically partitioning web applications in Swift . . . 2
- 1.3 The Fabric secure persistent object store . . . 2

2 Technical description **3**
- 2.1 Automatic security parameter inference . . . 3
- 2.2 Redesigning servlet containers for information flow . . . 3
- 2.3 Tracking information flow through a graphical user interface . . . 4
- 2.4 Application-defined principals . . . 4
- 2.5 Automatic policy-driven partitioning of web applications . . . 6
- 2.6 The Fabric secure persistent object store . . . 8
  - 2.6.1 Fabric architecture . . . 8
  - 2.6.2 Storage nodes . . . 10
  - 2.6.3 Worker nodes . . . 11
  - 2.6.4 Update maps for secure distributed computation . . . 11
  - 2.6.5 Distributed transaction management . . . 12
  - 2.6.6 Hierarchical commit protocol . . . 13
  - 2.6.7 Distributed secure transaction management . . . 14
  - 2.6.8 Type systems for reasoning about locality . . . 14

3 Results **15**
- 3.1 Software prototypes . . . 15
- 3.2 Publications . . . 15
- 3.3 Presentations . . . 16

4 Conclusions **17**

5 References **18**

6 List of symbols, abbreviations and acronyms **20**

List of Figures

1 Handling a request in SIF. 4
2 The Swift architecture 5
3 Fabric architecture 9
4 Logs of nested distributed transactions 12
5 A hierarchical, distributed transaction 13

1 Introduction

Web applications are now used for a wide range of important activities: email, social networking, on-line shopping and auctions, financial management, and many more. They provide services to millions of users and store information about and for them. However, a web application may contain design or implementation vulnerabilities that compromise the confidentiality, integrity, or availability of information manipulated by the application, with financial, legal, or ethical implications. According to a recent report [20], web applications account for 69% of Internet vulnerabilities. Prior techniques appear inadequate to prevent these vulnerabilities; better technology is needed for building web applications that are secure and that use information securely and accountably.

To enforce accountability of information, it is necessary to track and control how information flows through the system. Information flow control ensures that information affected by some source can be attributed to that source regardless of how indirect the effect is. Language-based methods can track information flow precisely, with fine granularity and low overhead.

In fact, information security vulnerabilities arise in general from inappropriate information dependencies, so tracking information flows within applications offers a comprehensive solution to security and accountability. Confidentiality can be enforced by controlling information flow from sensitive data to clients; integrity and accountability can be enforced by controlling information flow from clients to trusted information—as a side effect, protecting against common vulnerabilities like SQL injection and cross-site scripting. In fact, recent work [9, 12, 22, 10] on static analysis of web applications written in Java or PHP has used dependency analyses to find many vulnerabilities in existing web applications and web application libraries.

We have taken a language-based approach in which a compiler statically verifies the security of information flow. An alternative approach is to use purely dynamic tainting, which can detect some improper dependencies and has also proved useful in detecting vulnerabilities [24, 3]. However, static analyses have the advantage that they can conservatively identify information flows, including information flows through control flow, providing stronger security assurance [17]. In addition, our language-based approach is powerful enough that it can also track information flow dynamically when necessary.

Building applications that handle information securely is difficult. Analyzing the security of an application or system requires some assessment of whether the information used by the system is trustworthy, and to what degree. Without this assessment, trusted outputs of the system might be influenced by insufficiently trustworthy inputs or by the actions of insufficiently trustworthy agents. This assessment may be done as part of a validation process performed before the application is deployed; it may be also important to assess trustworthiness of information in running systems, because the provenance of information may inform decisions made using it.

1.1 The SIF web servlet architecture

Our first effort toward applying information flow controls to web applications was the Servlet Information Flow (SIF) servlet architecture [2]. This is a novel system for building high-assurance web applications. Assurance is provided by controlling information flow at the server side. Because all information provided to the web client is labeled with confidentiality and integrity, SIF exposes these security requirements automatically as part of the client-side user interface. Information flow analysis is known to be useful against attacks such as SQL injection and cross-site scripting, but SIF prevents inappropriate use of information more generally: flow of confidential information to clients is controlled, as is flow of low-integrity information from clients. Expressive policies allow users and application providers to protect information from each other.

SIF web applications are written in an extended version of the Jif (Java + information flow) programming language [13, 14], which itself extends Java with information-flow control. The enforcement mechanisms of SIF and Jif track the flow of information within a web application, and information sent to and returned from the client. SIF reduces the trust that must be placed in web applications, in exchange for trust in the servlet framework and the Jif compiler—a good bargain because the framework and compiler are shared by all SIF applications.

Language-based information flow promises cheap, strong information security. But until now, it could not effectively enforce information security in highly dynamic applications. To build SIF, we developed new language features that make it possible to write realistic web applications. Increased assurance is obtained with modest enforcement overhead.

1.2 Automatically partitioning web applications in Swift

Recent trends in web application design have exacerbated the security problem. To provide a rich, responsive user interface, application functionality is pushed into client-side JavaScript [6] code that executes within the web browser. JavaScript code is able to manipulate user interface components and can store information persistently on the client side by encoding it as cookies. These web applications are distributed applications, in which client- and server-side code exchange protocol messages represented as HTTP requests and responses. In addition, most browsers allow JavaScript code to issue its own HTTP requests, a functionality used in the Ajax (Asynchronous JavaScript and XML) development approach. Moving code and data to the client can create security vulnerabilities, but currently there are no good methods for deciding when it is secure to do so.

Swift is a new, principled approach to building web applications that are *secure by construction*. It automatically partitions application code while providing assurance that the resulting placement is secure and efficient. Application code is written as Java-like code annotated with information flow policies that specify the confidentiality and integrity of web application information. The compiler uses these policies to automatically partition the program into JavaScript code running in the browser, and Java code running on the server. To improve interactive performance, code and data are placed on the client side. However, security-critical code and data are always placed on the server. Code and data can also be replicated across the client and server, to obtain both security and performance. A max-flow algorithm is used to place code and data in a way that minimizes client–server communication.

The Swift prototype system has been released publicly and has been downloaded hundreds of time. A tutorial for programming in Swift has been developed. We have made it available on the Swift web site.

1.3 The Fabric secure persistent object store

SIF and Swift ensure that information flows are secure and that provenance of information is tracked via integrity policies. However, they do not handle information flows that pass through persistent storage. It is possible to use a Structured Query Language (SQL) database to store data persistently, as we have done with SIF and Swift applications. However, the application developer must manually reconstruct the security policies of information obtained from the database.

To address this lack, we developed Fabric, a secure persistent object store that allows distributed web applications to access persistent information securely in the form of objects. By building on top of Jif, the high-level programming language makes distribution and persistence largely transparent to programmers, while enforcing information flow policies on persistent objects. Fabric supports data-shipping and function-shipping styles of computation: both computation and

information can move between nodes to meet security requirements or to improve performance. Where current web applications might be structured to use queries, web applications based on Fabric can use remote computation.

Fabric provides a rich, Java-like object model, but data resources are labeled with confidentiality and integrity policies that are enforced through a combination of compile-time and runtime mechanisms. Optimistic, nested transactions ensure consistency across all objects and nodes. A peer-to-peer dissemination layer helps to increase availability and to balance load. Results from applications built using Fabric, include SIF-based web applications, suggest that Fabric has a clean, concise programming model, offers good performance, and enforces security.

2 Technical description

We now explore some of the technical innovations behind the SIF, Swift, and Fabric systems.

2.1 Automatic security parameter inference

To support programming in SIF, Swift, and Fabric, we developed a major extension to the Jif programming language, which supports automatic inference of many security annotations that previously were required. The motivation for this work is that even though Jif did already automatically infer most security annotations, the annotation burden was still too high, especially for the kind of code that appears in web applications. Our goal was to significantly reduce the annotation burden of building SIF applications.

Previously to this extension, it was typically necessary to supply security policy annotations when creating objects. For example, suppose that one wants to create a linked list of objects that are owned by `Alice` and readable by `Bob`. In Jif, it was previously necessary to add that annotation explicitly:

```
LinkedList[{Alice:Bob}] list = new LinkedList[{Alice:Bob}]();
```

With the new label inference mechanism, programmers can now write many fewer annotations. In this example, programmers can write the same expression they would in Java:

```
LinkedList list = new LinkedList();
```

The Jif compiler understands that the class `LinkedList` must be instantiated on a policy, and it automatically solves for that policy to determine that it can be `{Alice:Bob}`.

Inferring type parameters was a significant extension to the compiler. It is a kind of type inference that programming languages known for good type inference, such as ML, in fact do *not* do. One reason why parameter inference is usually avoided is that it can require expensive whole-program analyses that encourage brittle programs in which a small change to program can break faraway code. By contrast, the parameter inference features we added to Jif require only local analysis of method code. It is still necessary to annotate method signatures, but not the uses of classes parameterized on labels. We think this is constitutes a good "sweet spot" where the annotation burden is lessened without losing the good properties of the original Jif type inference algorithm.

2.2 Redesigning servlet containers for information flow

One of the challenges of the SIF project was how to design a web application servlet container to securely track information flow. We developed the architecture depicted in Figure 1.

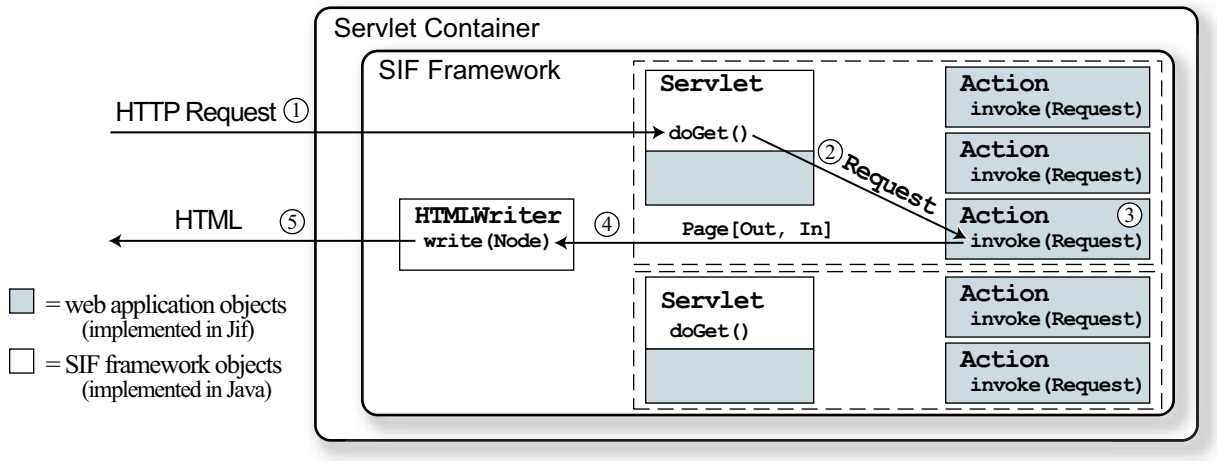

Figure 1: Handling a request in SIF.

The figure shows how an HTTP request is handled by SIF and the interaction with servlet code written in Jif.

1. An HTTP request is made from a web client to a servlet;
2. The HTTP request is wrapped in a `Request` object;
3. An appropriate `Action` object of the servlet is found to handle the request, and its `invoke` method called with the `Request` object;
4. The action's `invoke` method generates a `Page` object to return for the request;
5. The `Page` object is converted into HTML, which is returned to the client.

2.3 Tracking information flow through a graphical user interface

Another significant challenge posed by both the SIF and Swift frameworks was how to track information flow through a complex graphical user interface. This required developing a set of graphical user interface classes in which the possible information flows are described through type-level annotations.

These signatures make extensive use of powerful features of Jif. Classes representing HTML nodes are parameterized with respect to `In` and `Out` labels that constrain how information can flow through the tree of HTML widgets. The signatures make extensive use of dynamic labels [27] and dynamic principals [21], and many methods have `where` clauses that constrain labels and principals at compile time in complex ways.

While these features are complex, the payoff is that information flows are checked almost entirely at compile time. This means that the overhead of checking information flows is low, despite the richness of the security policy language. Further, it means that run-time failures do not happen, in contrast to purely dynamic taint-tracking schemes, where a bad information can cause the application to behave in unexpected (though secure) ways.

2.4 Application-defined principals

Principals are entities with security concerns. Applications may choose which entities to model as principals. Principals in Jif are represented at run time, and thus can be used as values by programs during execution. Jif gives run-time principals the primitive type `principal`. SIF

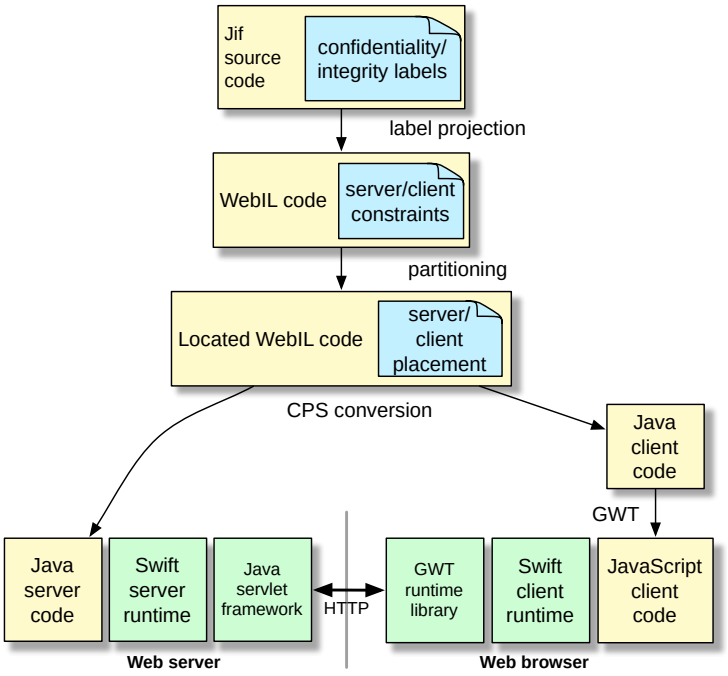

Figure 2: The Swift architecture

introduced an open-ended mechanism that allows applications great flexibility in defining and implementing their own principals.

Applications may implement the interface `jif.lang.Principal`. Any object that implements the `Principal` interface is a principal; it can be cast to the primitive type `principal`, and used just as any other principal. The `Principal` interface provides methods for principals to delegate their authority and to define authentication.

Delegation is crucial. For example, user principals must be able to delegate their authority to session principals, so that requests from users can be executed with their authority. The method call `p.delegatesTo(q)` returns `true` if and only if principal `p` delegates its authority to principal `q`. The implementation of a principal's `delegatesTo` method is the sole determiner of whether its authority is delegated. An *acts-for proof* is a sequence of principals $p_1, \ldots, p_n$, such that each p_i delegates its authority to p_{i+1}, and is thus a proof that p_n can act for p_1. Acts-for proofs are found using the methods `findProofUpTo` and `findProofDownTo` on the `Principal` interface, allowing an application to efficiently guide a proof search. Once an acts-for proof is found, it is verified using `delegatesTo`, cleanly separating proof search from proof verification.

The authority of principals is required for certain operations. For example, the authority of the principal $Alice$ is required to downgrade information labeled $\{Alice \rightarrow Bob \ ; \top \leftarrow \top\}$ to the label $\{Alice \rightarrow Bob, Chuck \ ; \top \leftarrow \top\}$ since a policy owned by $Alice$ is weakened. The authority of principals whose identity is known at compile time may be obtained by these principals approving the code that exercises their authority. However, for dynamic principals, whose identity is not known at compile time, a different mechanism is required. We extended Jif with a mechanism for dynamically authorizing closures.

2.5 Automatic policy-driven partitioning of web applications

The Swift system that we developed makes it possible to write applications that are *secure by construction*. Applications are written in a higher-level programming language in which information security requirements are explicitly exposed as declarative annotations. The compiler uses these security annotations to decide where code and data in the system can be placed securely, on the web server as Java or on the client browser as JavaScript. Code and data are partitioned at fine granularity, at the level of individual expressions and object fields. Developing programs in this way ensures that the resulting distributed application protects the confidentiality and integrity of information. The general enforcement of information integrity also guards against common vulnerabilities such as SQL injection and cross-site scripting.

Swift applications are not only more secure, they are also easier to write: control and data do not need to be explicitly transferred between client and server through the awkward extralinguistic mechanism of HTTP requests. Automatic placement has another benefit. In current practice, the programmer has no help designing the protocol or interfaces by which client and server code communicate. With Swift, the compiler automatically synthesizes secure, efficient interfaces for communication.

Of course, others have noticed that web applications are hard to make secure and awkward to write. Prior research has addressed security and expressiveness separately. One line of work has tried to make web applications more secure, through analysis [9, 22, 10] or monitoring [8, 15, 23] of server-side application code. However, this work does not help application developers decide when code and data can be placed on the client. Conversely, the awkwardness of programming web applications has motivated a second line of work toward a single, uniform language for writing distributed web applications [7, 4, 18, 26, 25]. However, this work largely ignores security; while the programmer controls code placement, nothing ensures the placement is secure.

The architecture of Swift is depicted in Figure 2. The system starts with annotated Java source code at the top of the diagram. Proceeding from top to bottom, a series of program transformations converts the code into a partitioned form shown at the bottom, with Java code running on the web server and JavaScript code running on the client web browser.

Jif source code The source language of the program is an extended version of the Jif 3.0 programming language [13, 14]. Jif extends the Java programming language with language-based mechanisms for information flow control and access control. Information security policies can be expressed directly within Jif programs, as *labels* on program variables. By statically checking a program, the Jif compiler ensures that these labels are consistent with flows of information in the program.

The original model of Jif security is that if a program passes compile-time static checking, and the program runs on a trustworthy platform, then the program will enforce the information security policies expressed as labels. For Swift, we assume that the web server can be trusted, but the client machine and browser may be buggy or malicious. Therefore, Swift must transform program code so that the application runs securely, even though it runs partly on the untrusted client.

WebIL intermediate code The first phase of program transformation converts Jif programs into code in an intermediate language we call Web Intermediate Language (WebIL). As in Jif, WebIL types can include annotations; however, the space of allowed annotations is much simpler, describing constraints on the possible locations of application code and data. For example, the annotation `S` means that the annotated code or data must be placed on the web server. The annotation `C?S` means that it must be placed on the server, and *may* optionally be replicated on the

client as well. WebIL is useful for web application programming in its own right, although it does not provide security assurance.

WebIL optimization The initial WebIL annotations are merely constraints on code and data placement. The second phase of compilation decides the exact placement and replication of code and data between the client and server, in accordance with these constraints. The system attempts to minimize the cost of the placement, in particular by avoiding unnecessary network messages. The minimization of the partitioning cost is expressed as an integer programming problem, and maximum flow methods are then used to find a good partitioning.

Splitting code Once code and data placements have been determined, the compiler transforms the original Java code into two Java programs, one representing server-side computation and the other, client-side computation. This is a fine-grained transformation. Different statements within the same method may run variously on the server and the client, and similarly with different fields of the same object. What appeared as sequential statements in the program source code may become separate code fragments on the client and server that invoke each other via network messages. Because control transfers become explicit messages, the transformation to two separate Java programs is similar to a conversion to *continuation-passing style* [16, 19].

JavaScript output Although our compiler generates Java code to run on the client, this Java code actually represents JavaScript code. The Google Web Toolkit (GWT) [7] is used to compile the Java code down to JavaScript. On the client, this code then uses the GWT run-time library and our own run-time support. On the server, the Java application code links against Swift's server-side run-time library, which in turn sits on top of the standard Java servlet framework.

The final application code generated by the compiler uses an Ajax approach to securely carry out the application described in the original source code. The application runs as JavaScript on the client browser, and issues its own HTTP requests to the web server, which responds with XML (Extensible Markup Language) data.

From the browser's perspective, the application runs as a single web page, with most user actions (e.g., clicking on buttons) handled by JavaScript code. This approach seems to be the current trend in web application design, replacing the older model in which a web application is associated with many different Uniform Resource Locators (URLs). One result of the change is that the browser "back" and "forward" buttons no longer have the originally intended effect on the web application, though this can be largely hidden, as is done in the GWT.

Partitioning and replication Compiling a Swift application puts some code and data onto the client. Code and data that implement the user interface clearly must reside on the client. Other code and data are placed on the client to avoid the latency of communicating with the server. With this approach, the web application can have a rich, highly responsive user interface that waits for server replies only when security demands that the server be involved.

In order to enforce the security requirements in the Jif source code, information flows between the client and the server must be strictly controlled. In particular, confidential information must not be sent to the client, and information received from the client cannot be trusted. The Swift compilation process generates code that satisfies these constraints.

One novel feature of Swift is its ability to selectively replicate computation onto *both* the client and server, improving both responsiveness and security. For example, validation of form inputs should happen on the client so the user does not have to wait for the server to respond when

invalid inputs are provided. However, client-side validation should not be trusted, so input validation must also be done on the server. In current practice, developers write separate validation code for the client and server, using different languages. This duplicates effort and makes it less likely that validation is done correctly and consistently. With Swift, the compiler can automatically replicate the same validation code onto both the server and the client. This replication is not a special-purpose mechanism; it is simply a result of applying a general-purpose algorithm for optimizing code placement.

2.6 The Fabric secure persistent object store

Web applications use persistent data, and for accountability it is crucial to be able to track the flow of information through persistent data. The Fabric system supports transparently persistent objects, and we showed that it could be used with SIF to provide comprehensive information flow tracking between the web application and the persistent store.

As we developed the Fabric persistence layer, we realized that in a world of distributed, replicated web applications, it was necessary to broaden the scope of this part of the project. Rather than just an object store, Fabric is a general way to manage persistent objects securely in a distributed system. Fabric provides a shared computational and storage substrate implemented by an essentially unbounded number of Internet hosts. As with the Web, there is no notion of an "instance" of Fabric. Two previously noninteracting sets of Fabric nodes can interact and share information without prior arrangement. There is no centralized control over admission: new nodes, even untrustworthy nodes, can join the system freely.

Fabric gives programmers a high-level programming abstraction in which security policies and some distributed computing features are explicitly visible to the programmer. Programmers access Fabric objects in a uniform way, even though the objects may be local or remote, persistent or nonpersistent, and object references may cross between Fabric nodes.

To achieve good performance while enforcing security, Fabric supports both *data shipping*, in which data moves to where computation is happening, and *function shipping*, in which computations move to where data resides. Data shipping enables Fabric nodes to compute using cached copies of remote objects, with good performance when the cache is populated. Function shipping enables computations to span multiple nodes. Inconsistency is prevented by performing all object updates within transactions, which are exposed at the language level. The availability of information, and scalability of Fabric, are increased by replicating objects within a peer-to-peer *dissemination layer*.

2.6.1 Fabric architecture

Fabric nodes take on one of the three roles depicted in Figure 3:

- *Storage nodes* (or *stores*) store objects persistently and provide object data when requested.
- *Worker nodes* perform computation, using both their own objects and possibly copies of objects from storage nodes or other worker nodes.
- *Dissemination nodes* provide copies of objects, giving worker nodes lower-latency access and offloading work from storage nodes.

Although Fabric nodes serve these three distinct roles, a single host machine can have multiple Fabric nodes on it, typically colocated in the same Java virtual machine. For example, a store can have a colocated worker, allowing the store to invoke code at the worker with low overhead. This

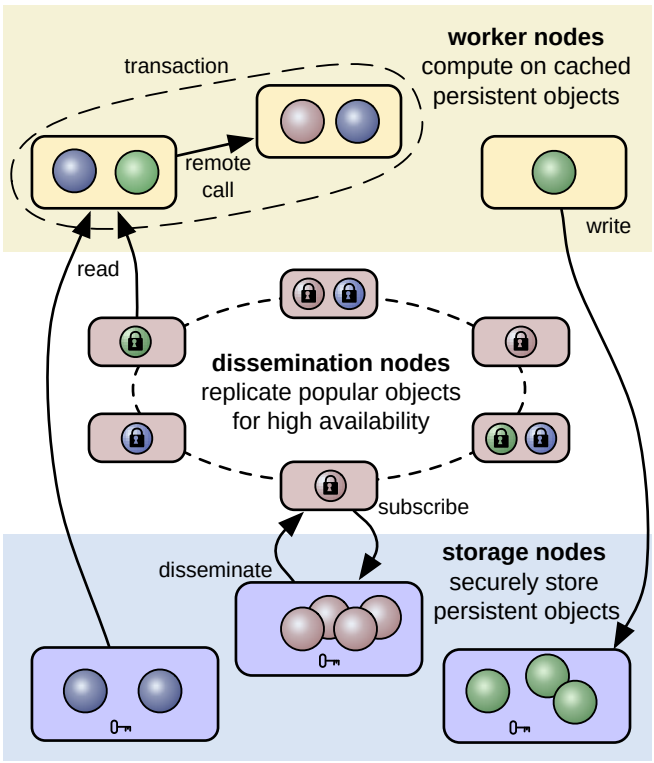

Figure 3: Fabric architecture

capability is useful, for example, when a store needs to evaluate a user-defined access control policy to decide whether an object update is allowed. It also gives the colocated worker the ability to efficiently execute queries against the store. Similarly, a worker node can be colocated with a dissemination node, making Fabric more scalable.

Object model Information in Fabric is stored in objects. Fabric objects are similar to Java objects; they are typically small and can be manipulated directly at the language level. Fabric also has array objects, to support larger data aggregates. Like Java objects, Fabric objects are mutable and are equipped with a notion of identity.

Naming Objects are named throughout Fabric by object identifiers (*oids*). An object identifier has two parts: a store identifier, which is a fully qualified Domain Name System (DNS) host name, and a 64-bit object number (*onum*), which identifies the object on that host. An object identifier can be transmitted through channels external to Fabric, by writing it as a URL with the form `fab://`*store*`/`*onum*, where *store* is the host name and *onum* is the object number.

An object identifier is permanent in the sense that it continues to refer to the same object for the lifetime of that object, and Fabric nodes always can use the identifier to find the object. If an object moves to a different store, acquiring an additional oid, the original oid still works because the original store has a surrogate object containing a forwarding pointer. Path compression is used to prevent long forwarding chains.

Knowing the oid of an object gives the power to name that object, but not the power to access it: oids are not capabilities [5]. If object names were capabilities, then knowing the name of an object would confer the power to access any object reachable from it. To prevent covert channels

that might arise because adversaries can see object identifiers, object numbers are generated by a cryptographically strong pseudorandom number generator. Therefore, an adversary cannot probe for the existence of a particular object, and an oid conveys no information other than the name of the node that persistently stores the object.

Fabric uses DNS to map hostnames to Internet Protocol (IP) addresses, but relies on X.509 certificates to verify the identity of the named hosts and to establish secure Secure Sockets Layer (SSL) connections to them. Therefore, certificate authorities are the roots of trust and naming, as in the Web.

Fabric applications can implement their own naming schemes using Fabric objects. For example, a naming scheme based on directories and pathnames is easy to implement using a hash map.

Labels Every object has an associated *label* that describes the confidentiality and integrity requirements associated with the object's data. It is used for information flow control and to control access to the object by Fabric nodes. This label is automatically computed by the Fabric run-time system based on programmer annotations and a combination of compile-time and run-time information flow analysis. Any program accepted by the Fabric type system is guaranteed to pass access control checks at stores, unless some revocation of trust has not yet propagated to the worker running it.

Classes Every Fabric object, including array objects, contains the oid of its *class object*, a Fabric object representing its class in the Fabric language. The class object contains both the fully-qualified path to its class (which need not be unique across Fabric) and the SHA-256 hash of the class's bytecode (which should be globally unique). The class object creates an unforgeable binding between each object and the correct code for implementing that object. The class object can also include the actual class bytecode, or the class bytecode can be obtained through an out-of-band mechanism and then checked against the hash. When objects are received over the network, the actual hash is verified against the expected one.

Versions Fabric objects can be mutable. Each object has a *current version number*, which is incremented when a transaction updates the object. The version number distinguishes current and old versions of objects. If worker nodes try to compute with out-of-date object versions, the transaction commit will fail and will be retried with the current versions. The version number is an information channel with the same confidentiality and integrity as the fields of the object; therefore, it is protected by the same mechanisms.

2.6.2 Storage nodes

Storage nodes (stores) persistently store objects and provide copies of object data on request to both worker nodes and dissemination nodes. Access control prevents nodes from obtaining data they should not see. When a worker requests a copy of an object from a store, the store examines the confidentiality part of the object's label, and provides the object only if the requesting node is trusted enough to read it. Therefore the object can be sent securely in plaintext between the two nodes (though it is of course encrypted by SSL). This access control mechanism works by treating each Fabric node as a principal. Each principal in Fabric keeps track of how much it trusts the nodes that it interacts with.

2.6.3 Worker nodes

Workers execute Fabric programs. Fabric programs may be written in the Fabric language. Trusted Fabric programs—that is, trusted by the worker on which they run—may incorporate code written in other languages, such as Fabric Intermediate Language (FabIL). However, workers will run code provided by other nodes only if the code is written in Fabric, and signed by a trusted node.

Fabric programs modify objects only inside transactions, which the Fabric programming language exposes to the programmer as a simple `atomic` construct. Transactions can be nested, which is important for making code compositional. During transactions, object updates are logged in an undo/redo log, and are rolled back if the transaction fails either because of inconsistency, deadlock, or an application-defined failure.

A Fabric program may be run entirely on a single worker that issues requests to stores (or to dissemination nodes) for objects that it needs. This data-shipping approach makes sense if the cost of moving data is small compared to the cost of computation, and if the objects' security policies permit the worker to compute using them.

When data shipping does not make sense, function shipping may be used instead. Execution of a Fabric program may be distributed across multiple workers, by using *remote method calls* to transfer control to other workers.

One important use of remote calls is to invoke an operation on a worker colocated with a store. Since a colocated worker has low-cost access to persistent objects, this can improve performance substantially. This idea is analogous to a conventional application issuing a database query for low-cost access to persistent data. In Fabric, a remote call to a worker that is colocated with a store can be used to achieve this goal, with two advantages compared to database queries: the worker can run arbitrary Fabric code, and information-flow security is enforced.

2.6.4 Update maps for secure distributed computation

Fabric supports shared access to persistent objects from multiple *worker nodes*. This is important for implementing web applications, where for throughput, multiple application servers must share access to the same underlying persistent information. Each app server is a worker node within the Fabric system. Worker node computations are structured as transactions. Each transaction runs in isolation from other Fabric transactions, and its side effects are committed atomically.

However, Fabric transactions can be distributed across multiple worker nodes by the use of remote calls within a transaction. The ability to distribute transactions is crucial for reconciling expressiveness with security. Although some workers are not trusted enough to read or write some objects, it is secure for them to perform these updates by calling code on a sufficiently trusted worker.

For consistency, workers need to see the latest versions of shared objects as they are updated. For performance, workers should be able to locally cache objects that are shared but not updated. For security, updates to an object with confidentiality L should not be learned by a worker $\mathtt{c}$ unless the labeling ordering $L \sqsubseteq \{\top \rightarrow \mathtt{c}\}$ holds (where $\mathtt{c}$ is a principal representing the degree of trust in the worker node).

These three requirements are reconciled by the use of *update maps*, which securely propagate object updates during the transaction. If an object is updated during a distributed transaction, the node performing the update becomes the object's *writer* and stores the definitive version of the object. If the object already has a writer, it is notified and relinquishes the role (this notification will not be a covert channel). The change of object writer is also recorded in the update map, which always resides at the single node that is currently executing in the transaction.

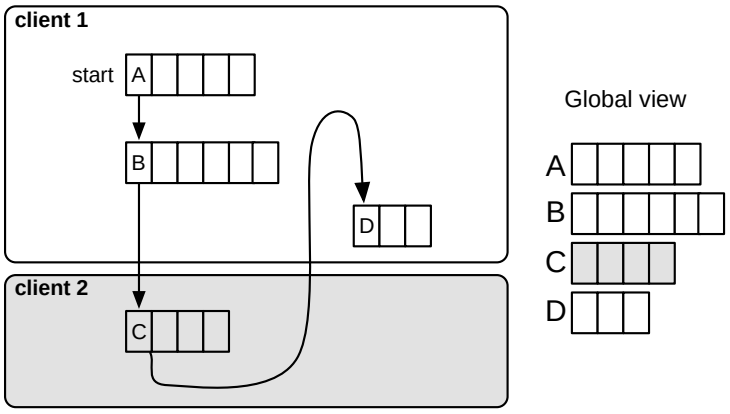

Figure 4: Logs of nested distributed transactions

The update map contains two kinds of mappings. An update to object `o` at worker `w` adds a mapping of the form $\texttt{MD5}(\texttt{oid}, \texttt{key}) \mapsto \mathit{enc}_{\texttt{key}}(\texttt{w})$, where `oid` is the oid of object `o`, `key` is its encryption key, *enc* represents symmetric-key encryption, and `MD5` represents use of the MD5 message digest algorithm. This mapping permits a worker that is about to read or write `o`—and therefore has the encryption key for `o`—to learn whether there is a corresponding entry in the update map, and to determine which node is currently the object's writer. The second kind of mapping supports object creation. The creation of a new object with oid `oid` adds an entry of the form $\texttt{MD5}(\texttt{oid}) \mapsto \texttt{oid}_{\texttt{label}}$, where $\texttt{oid}_{\texttt{label}}$ is the oid of the object's label, which contains the object's encryption key. This mapping allows a worker to find the encryption key for newly created objects, and then to check the update map for a mapping of the first kind.

Thus, the update map allows workers to efficiently check for updates to objects they are caching, without revealing information to workers that are not trusted to learn about updates.

2.6.5 Distributed transaction management

To maintain consistency, transaction management must in general span multiple workers. For each top-level transaction that a worker is involved in, it maintains transaction logs. These transaction logs must be stored on the workers where the logged actions occurred, because in general the logs may contain confidential information that other workers may not see. Figure 4 illustrates the log structures that could result in a distributed transaction involving two workers. In the figure, a transaction (A) starts on worker 1, then starts a nested subtransaction (B), then calls code on worker 2, which starts another subtransaction (C) there. That code then calls back to worker 1, starting a third subtransaction (D). Conceptually, all the transaction logs together form a single log that is distributed among the participating workers, as shown on the right-hand side. When D commits, its log is conceptually merged with the log of C, though no data is actually sent. When C commits, its log, including the log of D, is conceptually merged with that of B. In actuality, this causes the log of D to be merged with that of B, but the log for C remains on worker 2. When the top-level transaction commits, workers 1 and 2 communicate with the stores that have interacted with, using the logs.

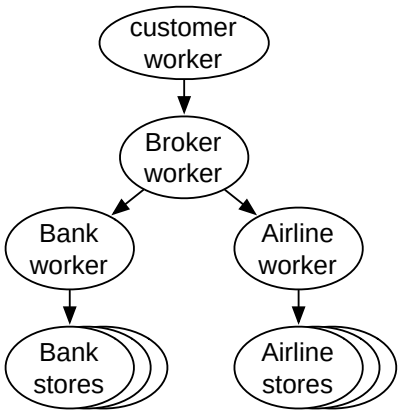

Figure 5: A hierarchical, distributed transaction

2.6.6 Hierarchical commit protocol

In general, a transaction may span worker nodes where there is mutual distrust. For example, consider a transaction that updates objects owned by a bank and other objects owned by an airline, perhaps as part of a transaction in which a ticket is purchased (see Figure 5). The bank and the airline do not necessarily trust each other; nor do they trust the customer purchasing the ticket. Therefore some computation is run on workers run respectively by the bank and the airline. When the transaction is to be committed, some updates to persistent objects are recorded on these different workers.

Because the airline and the bank do not trust the customer, it is necessary for the customer to find a trusted third party. Otherwise worker calls to update bank and airline data structures will be rejected because they lack sufficient integrity. As shown in the figure, a third-party broker can receive requests from the customer, use the Jif `endorse` mechanism to boost integrity, and then invoke operations on the bank and airline.

For security, Fabric commits transactions using a hierarchical version of two-phase commit. The worker initiates commit by contacting all the stores whose objects it has accessed, and all the other workers to which it has issued remote calls. These other workers then do the same. This procedure allows all the stores involved in a transaction to be informed about the transaction commit without relying on the top worker to choose which stores to contact and without revealing to the top worker which other workers and stores are involved in the transaction—which could be confidential. The two-phase commit protocol then proceeds as usual except that messages are passed up and down the call tree rather than directly between a single coordinator and the stores.

Of course, a worker in this tree could be compromised and fail to correctly carry out the protocol, causing some stores to be updated inconsistently with other stores. However, a worker that could do this could already have introduced this inconsistency by simply failing to update some objects or by failing to issue some remote method calls. In our example above, the broker could cause a ticket to be issued without payment being rendered, but only by violating the trust that was placed in it by the bank and airline. The customer's power over the transaction is merely to prevent it from happening at all, which is permitted.

Because workers act as coordinators, trust is placed in them to remain available, or to implement timely failure recovery, after the prepare phase of the transaction. Prepared transactions are timed out and aborted if the coordinator is unresponsive. In the example given, the broker

can cause inconsistent commits by permanently failing after telling only the airline to commit, in which case the bank will abort its part of the transaction. Of course, the broker already had that power. Thus, permanent failure is considered a violation of trust. To help prevent these permanent failures, the coordination of the second phase of the commit protocol could be replicated over a set of replicas running a consensus algorithm. We leave this refinement to future work.

Computations on workers run transactions optimistically, which means that a transaction can fail in various ways. The worker has enough information to roll the transaction back safely in each case. At commit time the system can detect inconsistencies that have arisen because another worker has updated an object accessed during the transaction. Another possibility is that the objects read by the transaction are already inconsistent, breaking invariants on which the code relies. Broken invariants can lead to errors in the execution of the program. Incorrectly computed results are not an issue because they will be detected and rolled back at commit time. Exceptions may also result; exceptions also cause transaction failure and rollback. Finally, a computation might diverge rather than terminate. Fabric handles divergence similarly to deadlocks; it times and retries transactions that are running too long. On retry, the transaction is given more time in case it it is genuinely a long-running transaction. By geometrically growing the retry time, the expected run time is only inflated by a constant factor.

2.6.7 Distributed secure transaction management

One of the challenges of the Fabric layer is how to handle transactions that go bad because they read inconsistent data. In most cases, this simply causes transaction commit to fail, and the transaction to be retried after fetching the latest versions of the stale objects. However, in some cases there is a danger of a failed transaction causing observable effects, violating consistency and possibly leaking sensitive information. There are two ways that inconsistency is observable. The first is a transaction that goes into an infinite loop, and thus never tries to commit. The second way is a transaction that generates an exception. To our knowledge these problem with optimistic transactions have not been resolved in the past. Our solution to nontermination is to automatically abort and retry long-running transactions with a geometrically increasing series of timeouts. This strategy ensures that a correct transaction will eventually complete with only a constant-factor end-to-end slowdown. To deal with exceptions, we consume all nonfatal exceptions and retry the transaction. In both cases the objects read by the transaction up to the point of retry can be identified from the transaction log. Up-to-date versions of the objects are obtained, avoiding falling into the same inconsistency again.

2.6.8 Type systems for reasoning about locality

Persistent objects pose some new challenges from the standpoint of building reliable, secure systems. In particular, we want Fabric to prevent deletion of persistent objects, which would violate integrity guarantees. Therefore, Fabric implements a policy of keeping *reachable* objects persistent. This strategy has been used by object-oriented databases in the past, but in the context of Fabric, it has some weaknesses that we have been addressing by adding new type-level annotations that capture *persistence* policies. In the language of ACID (atomicity, consistency, isolation, and durability) properties provided by databases, these policies relate to *durability*. However, language-level policies for persistence appear to be a completely novel idea.

One classic weakness of persistence-by-reachability is that objects can become persistent *accidentally*, using up persistent storage space. In the context of Fabric, where untrusted data can affect which pointers are created by possibly buggy code, we also need to worry about *malicious*

attacks on storage space. We have extended the Jif label model with a new policy language that controls both accidental and malicious persistence. Accidental persistence is controlled by associating persistence levels with objects that place bounds on how persistent an object will be; malicious persistence is controlled by preventing untrusted code from creating strong references to objects.

We have formalized the new policies and the type system of this simplified language, and have formalized the security properties that this type system should enforce. We have left integration of these new persistence policies into the Fabric programming language for future work.

3 Results

3.1 Software prototypes

We have built working prototypes of all the systems described in this report, and evaluated their performance and usability on a variety of web applications.

We have released a prototype of the SIF (Servlet Information Flow) web application framework, along with prototype versions of various useful applications, including a multi-user shared calendar and a Cross-Domain Information Sharing (CDIS) email application. Performance evaluation of the SIF framework showed that the performance loss due to run-time checking of labels and principals was modest, even when compared to a carefully engineered Java-based web application [2].

We have also released a prototype of the Swift web application partitioning framework, including a variety of different interactive example applications. The Swift prototype comes along with a tutorial that walks the reader through the construction of a simple web secure web application. We evaluated the Swift prototype on a variety of web applications, and showed that the compiler was able to partition web applications in a way that avoided extra network communication and that minimized latency [1].

A working prototype of the Fabric platform has been developed, incorporating the features described in this report. We have done a preliminary evaluation of this system, focusing on expressive power and performance. One important evaluation of Fabric has been to port a 50,000 line course management system (CMS), originally written in Java, to run on top of Fabric. The performance results are encouraging: the Fabric version of CMS runs several times faster than the production version that is built using the industry-standard approach of Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) layered over an Microsoft SQL Server or Oracle database—while offering strong security enforcement [11]. Because Fabric supports serializable transactions, it is very easy to replicate application servers, scaling up the performance of the web application without harming security or consistency. We showed that scalability was easily achieved.

The goal of the Fabric persistence layer is to support persistence for Jif-based web applications. We have integrated the SIF web application framework with Fabric, so that SIF web applications can now store persistent objects directly into Fabric, while automatically preserving confidentiality and integrity policies defined by the web application. We also modified the SIF multiuser calendar application to use Fabric instead of using the less-secure SQL database it was originally designed for.

3.2 Publications

Work on the following refereed publications was supported by this award:

1. Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. *Proc. ACM Symposium on*

Operating Systems Principles (SOSP). October 2009.

2. Xin Qi and Andrew C. Myers. Sharing classes between families. *Proc. ACM Conference on Programming Language Design and Implementation* (PLDI), pp. 281–292, June 2009.

3. Xin Qi and Andrew C. Myers. Masked types for sound object initialization. *Proc. ACM Symposium on Principles of Programming Languages* (POPL), January 2009.

4. Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proc. 2008 IEEE Computer Security Foundations Symposium*, pp. 98–111, June 2008.

5. Lantian Zheng and Andrew C. Myers. Securing nonintrusive web encryption through information flow. In *Proc. 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 125–134, June 2008.

6. Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proc. IEEE Symposium on Security and Privacy*, pp. 354–368, May 2008.

The following publication was invited to the Communications of the ACM:

7. Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Building secure web applications with automatic partitioning. *Comm. of the ACM*, (2), 2009.

Additional papers supported by this award are in submission to a peer-reviewed conference:

8. Aslan Askarov and Andrew C. Myers. A semantic framework for declassification and endorsement policies. Submitted to *2010 European Symposium on Programming*, March 2010.

9. Xin Qi and Andrew C. Myers. Homogeneous family sharing. Submitted to *2010 European Symposium on Programming*, March 2010.

3.3 Presentations

This award supported travel to a number of meetings and presentations related to the research topic:

1. NICIAR PI meeting, Chicago, Illinois, April 7–9, 2008. Attendees: Myers. Presented a poster and talk on this project.

2. IEEE Symposium on Security and Privacy. Berkeley, California, May 18–21, 2008. Attendees: Andrew Myers, Stephen Chong, Michael Clarkson. Presented the paper *Civitas: Toward a secure voting system.*

3. STONESOUP software assurance workshop, meeting 1. BWI Airport, May 9, 2008. PI Myers headed an IARPA study on new approaches for obtaining software assurance in place of the existing evaluation and certification process. This meeting focused on operating-system-level mechanisms for security enforcement.
`http://www.cs.cornell.edu/andru/stonesoup/`. Attendees: Myers.

4. "Guiding distributed systems synthesis with language-based security policies." Invited talk, 10th IFIP International Conference on Formal Methods for Open Object-Oriented Distributed Systems (FMOODS'08). Oslo, Norway, June 4, 2008.

5. STONESOUP software assurance workshop, meeting 2. Microsoft Research, Redmond, Washington, June 20, 2008.
   `http://www.cs.cornell.edu/andru/stonesoup/`. Attendees from this project: Myers.

6. STONESOUP software assurance workshop, meeting 3. Arlington, Virginia, August 6, 2008. This meeting focused on producing the final briefing and getting input from DoD and NSA representatives.
   `http://www.cs.cornell.edu/andru/stonesoup/`. Attendees from this project: Myers.

7. STONESOUP final briefing. College Park, MD, October 24, 2008. Attendees from this project: Myers.
   `http://www.cs.cornell.edu/andru/stonesoup/`. Attendees from this project: Myers.

8. ACM Symposium on Principles of Programming Languages (POPL 2009). A premier conference on programming languages. January 21–23, Hyatt Regency, Savannah, Georgia. Attendees: Xin Qi, Andrew Myers. Presentation: "Masked types for sound object initialization."

9. 2009 IEEE Security and Privacy Program Committee Meeting. Andrew Myers was the 2009 Program Committee Chair for this annual conference. January 26–27, University of Maryland, College Park, Maryland. Attendees: Andrew Myers.

10. IBM PL Day, May 7, Hawthorne, New York. Presented keynote talk on the Fabric secure persistence layer: "Fabric: a higher-level abstraction for secure distributed programming."

11. 2009 IEEE Symposium on Security and Privacy. May 18–20, Claremont Resort, Oakland, California. Andrew Myers was the Program Committee Chair for this annual conference.

12. 2009 ACM Symposium on Operating Systems Principles (SOSP) Program Committee Meeting, June 4, University of Washington, Seattle, Washington.

13. 2009 SOSP Program Committee Workshop, June 5, University of Washington, Seattle, Washington. Presented a talk on our work on secure voting: "Civitas: a secure remote voting system".

14. 2009 ACM Conference on Programming Language Design and Implementation (PLDI). June 16–18, Dublin, Ireland. Xin Qi presented our paper on family sharing for extensibility.

4 Conclusions

This research has explored a new approach to building secure web applications. The unifying idea is to give the programmer a common, high-level abstraction for programming web applications. In this abstraction, programmers can express the key security requirements of these applications declaratively through labels for confidentiality and integrity of information. These labels ensure accountability by ensuring that security policies must be enforced throughout.

In the Swift system, we showed that programmers could write high-level code for both the server side and client side of a web application, and a compiler could automatically partition the code securely and efficiently. This unified the top two tiers of a web application in a common abstraction that enforced security.

In the Fabric system, we concentrated on the bottom two tiers of a web application instead. By running SIF on top of Fabric, we showed that a web application could manage persistent data objects conveniently, securely, and efficiently. Again, the key was to have a common security policy framework, based on information flow control, that spanned both tiers of the application.

There is much more that can be done to extend this work. An obvious next step would be to combine the Fabric and Swift approaches, allowing an entire three-tier web application to be implemented with single language-level abstraction, while providing distribution, consistency, persistence, and strong security enforcement. We leave this integration to future work.

Our experience with programming in these various systems suggests that the annotation burden is still relatively high, despite improvements in policy inference that were developed during the project. Another fruitful direction would be to develop better inference methods and better support for programmer diagnosis of information flow violations.

5 References

[1] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, October 2007.

[2] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symposium*, August 2007.

[3] Lap chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. 22st Annual Computer Security Applications Conference (ACSAC 2006)*, December 2006.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, November 2006.

[5] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, March 1966.

[6] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, 2002.

[7] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

[8] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.

[9] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proc. 13th International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy*, pages 258–263, May 2006.

[11] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.

[12] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. USENIX Annual Technical Conference*, pages 271–286, August 2005.

[13] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[14] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[15] A. Nguyen-Tuong, S. Guarneri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th International Information Security Conference*, pages 372–382, May 2005.

[16] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

[17] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[18] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proc. 1st Dynamic Languages Symposium*, pages 975–985, October 2006.

[19] Guy L. Steele, Jr. RABBIT: A compiler for Scheme. Technical Report AITR-474, MIT AI Laboratory, Cambridge, MA, May 1978.

[20] Symantec Internet security threat report, volume IX. Symantec Corporation, March 2006.

[21] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[22] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symposium*, pages 179–192, July 2006.

[23] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. 15th USENIX Security Symposium*, pages 121–136, August 2006.

[24] Wei Xu, V.N. Venkatakrishnan, R. Sekar, and I.V. Ramakrishnan. A framework for building privacy-conscious composite web services. In *4th IEEE International Conference on Web Services (ICWS'06)*, September 2006.

[25] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applictions with automatic client-server partitioning. In *Proc. 16th International World Wide Web Conference (WWW'07)*, pages 341–350, 2007.

[26] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. 22nd International Conference on Data Engineering (ICDE'06)*, pages 32–43, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

6 List of symbols, abbreviations and acronyms

ACID Atomicity, Consistency, Isolation, Durability
Ajax Asynchronous JavaScript and XML
CMS Course Management System
DNS Domain Name System
CDIS Cross-Domain Information Sharing
GWT Google Web Toolkit
FabIL Fabric Intermediate Language
J2EE Java 2 Enterprise Edition
EJB Enterprise JavaBeans
Jif Java + information flow
JVM Java Virtual Machine
HTTP Hypertext Transfer Protocol
HTML Hypertext Markup Language
IP Internet Protocol
NFS Network File System
oid Object identifier
onum Object number
PHP PHP: Hypertext Processor
SIF Servlet Information Flow
SSL Secure Sockets Layer
SHA Secure Hash Algorithm
SQL Structured Query Language
URL Uniform Resource Locator
WebIL Web Intermediate Language
XML Extensible Markup Language